Software Engineering Institute

Architecting Service-Oriented Systems

Philip Bianco
Grace A. Lewis
Paulo Merson
Soumya Simanta

August 2011

TECHNICAL NOTE
CMU/SEI-2011-TN-008

Research, Technology, and System Solutions Program

http://www.sei.cmu.edu

Carnegie Mellon

This material is based upon work supported by United States Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

This report was prepared for the

SEI Administrative Agent
ESC/XPK
5 Eglin Street
Hanscom AFB, MA 01731-2100

Table of Contents

List of Figures

List of Tables

Abstract

Service orientation is an approach to software systems development that has become a popular way to implement distributed, loosely coupled systems, because it offers such features as standardization, platform independence, well-defined interfaces, and tool support that enables legacy system integration. From a quality attribute point of view, the primary drivers for service orientation adoption are interoperability and modifiability. However, a common misconception is that an architecture that uses a service-oriented approach can achieve these qualities by simply putting together a set of vendor products that provide an infrastructure and then using this infrastructure to expose a set of reusable services to build systems. In reality, there are many architectural decisions that need to be made. An architectural decision that promotes interoperability or modifiability can negatively impact other qualities, such as availability, reliability, security, and performance. The goal of this report is to present general guidelines for architecting service-oriented systems, how common service-oriented system components support these principles, and the effect that these principles and their implementation have on system quality attributes.

1 Introduction

Despite a highly publicized report that claimed that "SOA is Dead,[1]" the reality is that service-oriented architecture (SOA) is still a popular architectural style for designing and developing distributed systems. As with any architectural style, SOA can be described in terms of the important architectural elements and the relationships among them. In this report, we examine how the design of these elements and their relationships impact system quality.

Solutions that use a service-oriented[2] approach are intended to satisfy business or mission goals that include quality requirements such as easy and flexible integration with legacy systems (interoperability), streamlined business processes (maintainability), reduced costs (modifiability), and agility to handle rapidly changing business processes (extensibility). These are the primary architectural drivers addressed by SOA adoption, and are achieved by adhering to a set of design principles for service-oriented systems that will be described later in the report. However, there are other important quality attributes such as availability, reliability, security, and performance that have to be addressed. In addition, an architectural decision that promotes one of these quality attributes can negatively impact any other quality attribute.

As an architectural pattern, SOA is an appropriate solution in some situations; however, there are situations in which it is not appropriate or in which it has to be used in conjunction with other technologies to achieve desired system qualities. A few examples of situations when SOA may not be appropriate include the following:

- In a solution that does not require the integration of components or systems running on different platforms, or implemented using different technologies, service orientation may be overkill because there is an overhead for the use of SOA technologies to provide interoperability across platforms.
- For a system built on a homogenous development platform with few interactions with legacy systems running on different platforms in which the situation is not likely to change, a move towards service orientation is hard to justify.
- If a system consists of co-located components or distributed components that interact via email messages, file sharing or proprietary messaging systems, the sophisticated integration mechanisms provided by SOA may introduce an unnecessary burden.
- Hard real-time systems are clearly not a good match for service orientation. Strict timeliness requirements conflict with several aspects of common technologies used in service-oriented systems (e.g., web services) that may introduce unbounded overhead in processing, such as extensible markup language (XML) parsing, validation, and transformation; network com-

[1] http://apsblog.burtongroup.com/2009/01/soa-is-dead-long-live-services/comments/page/2/

[2] We use the term *service-oriented system* to separate SOA as a set of technologies from service-orientation as a system or approach that incorporates and applies SOA-related concepts and technologies.

munication; proxies and stubs for technology adaptation; and intermediary components (e.g., a service registry or an enterprise service bus [ESB]).

- Embedded systems are not naturally fit to host service-oriented systems. Embedded platforms have limited computing power, memory, and disk resources. Many SOA technologies are heavyweight in terms of memory and CPU requirements. Thus, designing a SOA solution for the software on a washing machine or a video-game console can bring unneeded complication and overhead.

Architects therefore play a crucial role in determining what expectations can or cannot be met by SOA adoption, and where decisions can be made for the benefit of the organization and the accomplishment of system quality attributes. Reasoning about these difficult decisions can be simplified by using known solutions for promoting quality attributes that are important to the systems' stakeholders. These known solutions are often codified as architectural patterns and tactics. An architectural pattern "deals with a specific, recurring problem in the design of a software system…to construct architectures with specific properties [Buschmann 1996]." Architectural patterns are used to generate designs that are predictable and well understood. Architectural patterns can be decomposed into a set of architectural tactics. Architectural tactics are design decisions that are known to influence quality attribute responses [Bass 2003]. An example of a tactic is to introduce redundancy to promote system availability by reducing system downtime (e.g., system availability rises from 99.0% to 99.9% when a redundant element is added).

The goal of this report is to show architects of service-oriented systems how to decompose these systems into a set of architectural patterns and tactics that promote important system quality goals. Section 2 summarizes existing related work. Section 3 presents a set of SOA architectural principles that are realized through patterns and tactics. Section 4 presents the common elements of a service-oriented system, how these elements support the SOA architectural principles, and the system qualities that these elements promote.

2 Summary of Existing Work

A common misconception (mostly vendor driven) is that simply by adopting a SOA strategy or even acquiring a SOA infrastructure, an organization has established a complete well-crafted architecture that will help the organization achieve its many business goals [Lewis 2007]. In reality, SOA is an architectural pattern from which an infinite number of architectures can be derived—both good and bad. Appropriate decisions regarding tradeoffs are very specific to the system in consideration, providing one more reason why organizations make a mistake in assuming that SOA represents a "finished" architecture. This section summarizes existing work that addresses the architecture and design of service-oriented systems as a key activity in the implementation of service-oriented systems. Our report builds on this existing work to provide guidance for architects that need to make design decisions in service-oriented systems.

2.1 SOA Design Patterns

Architectural patterns are used to generate designs that are predictable and well understood. These patterns leverage knowledge and experience to produce proven solutions to recurring design problems. The book *SOA Design Patterns* by Thomas Erl (with contributions from over thirty practitioners) as well as the SOA Patterns website, describe approximately eighty-five patterns for service-oriented systems. The goal of SOA design patterns is to provide a "master catalog and pattern language for SOA" for practitioners that are designing a system using service orientation [Erl 2009]. Some of these patterns have been described in other work, such as *Design Patterns* and *Pattern-Oriented Software Architecture* [Gamma 1994, Buschmann 1996]. Thomas Erl shows how these patterns relate to the principles of service-oriented design. Examples of patterns include

- Patterns for the infrastructure needed to support service orientation, such as *Enterprise Service Bus (ESB)* and *Service Registry*
- Patterns for creating inventories of services, such as *Service Normalization* and *Service Layers*
- Patterns for composing services, such as *Entity Abstraction*, *Agnostic Context*, *Capability Composition*, and *Enterprise Inventory*
- Patterns for reliable messaging, such as *Reliable Messaging*
- Patterns for atomic distributed service transactions, such as *Atomic Service Transaction*
- Patterns for security, such as *Brokered Authentication*, *Data Origin Authentication* and *Data Confidentiality*
- Patterns for encapsulating legacy systems, such as *Legacy Wrapper* and *Service Messaging*

2.2 Evaluating SOA

Evaluating the architecture of a service-oriented system is not much different from evaluating any other kind of software architecture. The goal is to assess the ability of the software architecture to successfully address the requirements of the system or, more broadly, the business goals. The basic principles for the evaluation of any software architecture using a business-goal and quality-

attribute-based approach, such as the Architecture Tradeoff Analysis Method® (ATAM®) [Bass 2003], include

- Quality attribute requirements shape the architecture. If achievement of functionality were the only requirement, the system could exist as a single monolithic module with no internal structure at all [Bass 2003]. However, components and infrastructure elements are replicated because of availability requirements, concurrency is introduced, network and database access are avoided because of performance requirements, modules are organized into layers because of modifiability requirements, and so on. Therefore, a software architecture can only be evaluated if the quality attribute requirements are understood.
- A broad group of stakeholders need to be involved in the elicitation of quality requirements because they bring together a variety of disparate concerns that the architect might overlook. Network administrators may have specific requirements for bandwidth utilization; information security personnel may bring new confidentiality requirements; the database administrator may want to constrain the execution time of data access operations; an external service consumer may have interoperability requirements; and so on.
- Design decisions often incur quality attribute tradeoffs. In a software architecture evaluation, the appropriateness of each design decision is weighed only after the importance of each quality attribute requirement is understood.
- Because architectural decisions tend to have a pervasive effect on implementations and have a significant impact on business, performing an early architecture evaluation is particularly valuable and recommended.
- When a system includes connectivity with other systems and business entities, the political forces involved with this connectivity can be as important as both technical and non-technical concerns in architecture tradeoff considerations.

These architectural evaluation principles can be applied to service-oriented systems because these systems are often part of technologically diverse environments that involve a large number of design considerations. Examples of SOA-related design decisions that are explored during an evaluation and should be considered during the design process include

- What communication protocol should be used between each pair of service consumer and service provider?
- What integration approach should be followed? ESB-based or direct point-to-point?
- Should an orchestration server (e.g., a business process execution language [BPEL] engine) be used?
- Should a service registry be used? What capabilities besides naming and location would it provide?
- Should a defined service operation be synchronous or asynchronous?
- What is the appropriate granularity for the operations in each service interface?
- What are the system strategies for exception handling and fault recovery?

® Architecture Tradeoff Analysis Method and ATAM are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

- Should message-level security be used to protect messages? Or is channel-level security enough?
- What authentication and authorization mechanisms will be used?
- What service versioning mechanism will be used?

Evaluating a Service-Oriented Architecture [Bianco 2007] discusses the design decisions listed above and several others, but also provides the pros and cons of the different design alternatives with respect to various quality attributes. For example, static service binding (with no registry) yields better response time, whereas dynamic service binding (with a registry) incurs a performance overhead but yields better modifiability. The report also lists sample design questions that could be raised in an architecture evaluation and during the design of a service-oriented system to make sure that different quality requirements are addressed. Some examples of these questions are

- Which standards does the supporting platform implement?
- Can this particular service operation be called asynchronously?
- Do operations in the service interfaces map to transactional boundaries?
- How stable is the business process that will be executed in the service-oriented system?
- Which types of failures is the system subject to?
- Does the architecture provide a mechanism (e.g., digital signatures) to ensure that a third party will not intercept and alter message contents (tampering)?
- What kind (e.g., lightweight directory access protocol [LDAP]-based) and scope (e.g., enterprise-wide) of security domain are going to be used for managing the identity of participating users and systems?
- Can XML validation be turned off?
- Is a registry being used? If so, is it used for dynamic routing of service calls (e.g., for failover)?
- If an orchestration engine is being used, does it generate audit trails that support transaction traceability and regulatory requirements?
- How long should old versions of services/operations be available?
- What is the unit of versioning? Service or operation within a service?

2.3 SOA Layers

There are many sources that provide a form of reference architecture or layered approach for systems that use a service-orientation approach [Bieberstein 2008, Arsanjani 2004]. These layers facilitate separation of concerns, and designers have a set of architectural decisions that need to be made in each layer. Figure 1 shows the typical layers of a service-oriented system that are primarily functional in nature.

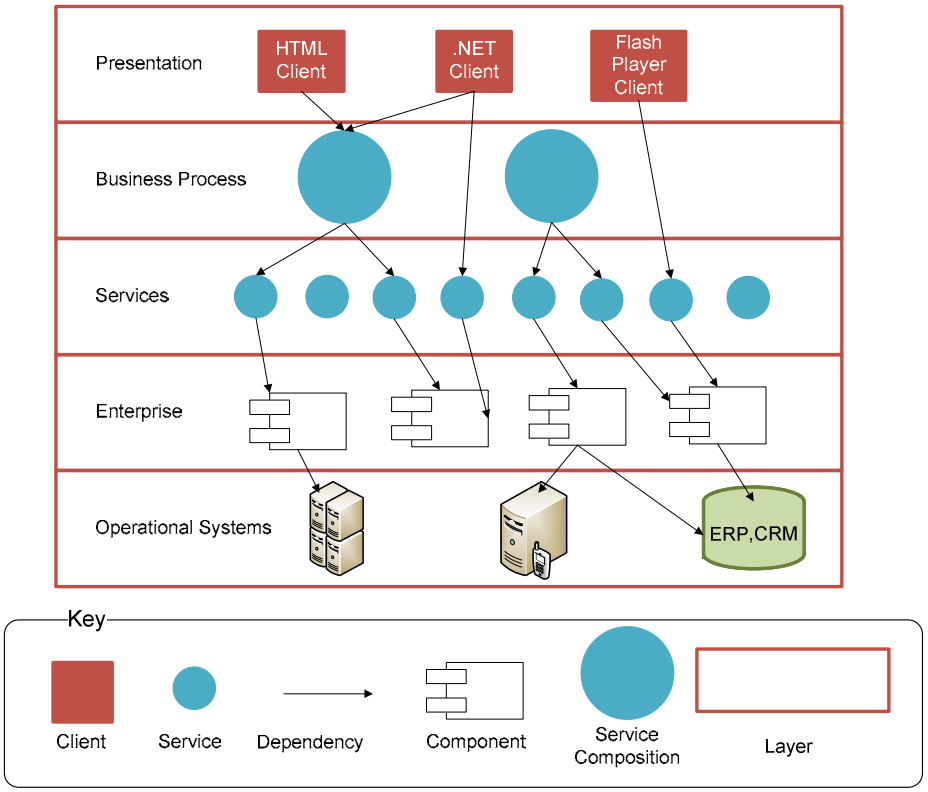

Figure 1: SOA Layers

This service-oriented system includes the following layers:

- Presentation—The benefit of creating a presentation layer is decoupling the client-side presentation implementation from the service implementation to allow each to change independently. This layer is generally not the focus of service-oriented design.
- Business process—The services provided in the services layer are often composed into workflows to assist in the development of applications. This provides flexibility to change the workflows as business processes change.
- Services—Services reside in this layer. The invocation of these services can be determined at design time or bound dynamically through a service registry at runtime. These services are broken into additional layers to assist in the composition of services into complete business processes. The additional layers include:
  - Utility-based services, which provide utility-based functions such as notification, logging and exception handling. These operations are largely agnostic to business processes and can therefore be reused in multiple business processes.
  - Entity-based services, which operate on a set of business entities (or data entities). These operations are largely agnostic to business processes and can therefore be reused in multiple business processes.
  - Task-based services, which are “business services with a functional boundary directly associated with a specific parent business task or process” [SOA Methodology 2010].

- Enterprise—These components contain code that specifically fulfills service needs or code that accesses the functionality in operational systems. “These special components are a managed, governed set of enterprise assets that are funded at the enterprise or the business unit level. …This layer typically uses container-based technologies such as application servers to implement the components, workload management, high-availability, and load balancing” [IBM 2004].
- Operational Systems—This layer consists of existing custom-built, commercial, external systems or some combination of these. Careful analysis of these systems needs to be completed to determine the operations that should be exposed as services. These services are considered to have enterprise-wide utility.

Many reference architectures provide additional layers for integration, governance, monitoring, and management, as will be discussed in Section 4.5.

3 SOA Architectural Principles

SOA architectural principles are general guidelines for architecting service-oriented systems. These principles are ideally enabled by the decisions found in the architecture of the system. In a service-oriented architectural pattern we characterize explicit boundaries between its four main types of elements: service consumers, SOA infrastructure, service interfaces, and service implementation, as shown in Figure 2.[3]

Erl and others have defined additional principles for service design [Erl 2008]. The principles in this section are similar, but they apply to the full architecture of the service-oriented system: the integration of services (interface and implementation), service consumers, and the SOA infrastructure. Each principle contains a short description and a table that explains the effects that each principle has on selected system quality attributes.

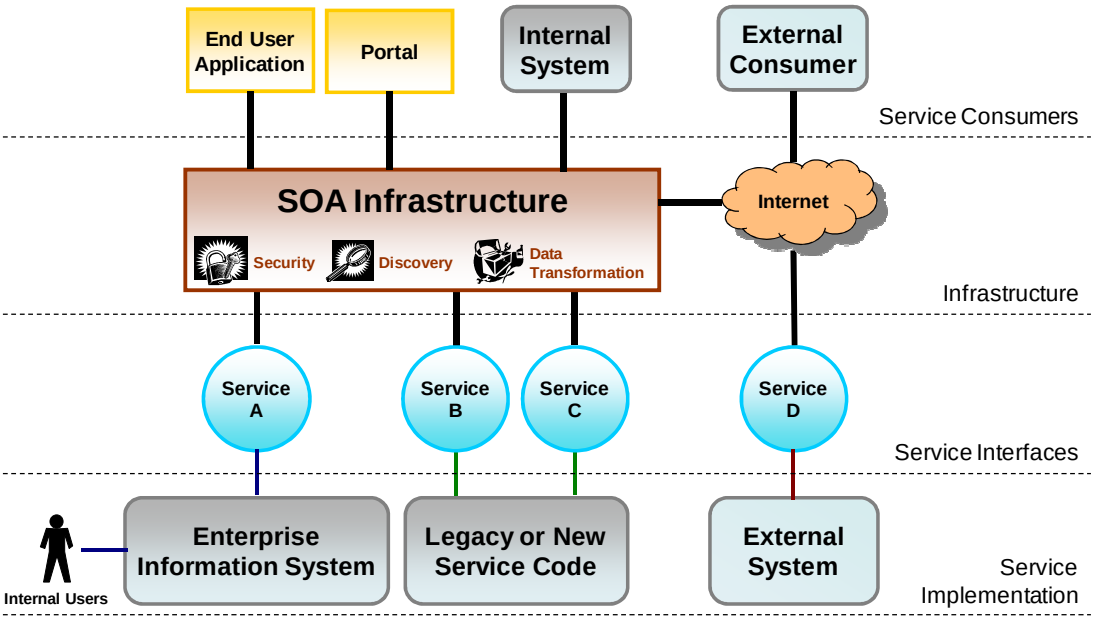

Figure 2: High-Level Notional View of a Service-Oriented System

Architects of service-oriented systems often find themselves in a conflict. On one hand, there are business/mission goals and quality attribute requirements driving the architecture of a system. On the other hand, there are principles of service-orientation that influence the architecture of a system and impact a system's quality attributes. It is at this intersection of these two sets of quality attributes where conflicts arise and an architect needs to make decisions. The responsibility of the architect is to try to apply each principle in the context of the business goals of the system and to make the necessary tradeoffs and architectural decisions in order to meet the system's business

[3] Figure 2 is a much more generic representation of the elements shown in Figure 1. The goal of Figure 2 is to illustrate the four major elements of a service-oriented system, independent of implementation technologies.

goals. It is important to note that the impact on quality is not binary (positive or negative) because specific system context may impact the effect on quality attributes of interest. The information contained in each of the following subsections reflects general trends.

3.1 Standardization (Interoperability)

One of the enablers of widespread SOA adoption, especially in the case of WS* web services,[4] is standardization at multiple levels, as shown in Figure 3. Standardization in service-oriented systems has multiple advantages including tool support and leverage of third-party system components that in the end can lead to shorter development times.

The WS* base stack (HTTP, XML, SOAP, and WSDL) is fairly stable and has large tool support. For example, there are multiple tools that will take a web service definition language (WSDL) document as input and produce all the code necessary to invoke the associated service. However, beyond the base stack it is not that straightforward because of the over-abundance of standards. There are currently over 100 WS* standards produced by organizations such as OASIS and W3C in areas that include business process specification, composition, messaging, reliable messaging, transaction management, security, and management. Some of these standards are complementary and some are competing. In addition, many of these standards have extensions, as well as areas that can be interpreted in multiple ways [Lewis 2008a].

The Web Services Interoperability Organization (WS-I) is an organization chartered to promote web services interoperability across platforms, applications, and programming languages [WS-I 2010]. WS-I has profiles for the basic stack and for security to provide clarifications, refinements, interpretations, and amplifications in areas of the standards that are subject to multiple interpretations. There are also tools to check that artifacts (e.g., a WSDL file) and actual messages being exchanged are in conformance with the profiles. The WS-I tools are especially useful in cases in which WSDL and XML files are automatically generated and may not conform to the assumptions of the development and deployment environment, e.g., different XML schema versions, different namespaces, malformed XML, etc. The tools that facilitate the usage of these standards will be one criterion that an architect uses to select between competing standards.

[4] In WS-* Web services, (1) data is represented using XML, (2) service interfaces are described using Web Services Description Language (WSDL), (3) payload is transmitted using Simple Object Access Protocol (SOAP) over Hypertext Transfer Protocol (HTTP), and, optionally, (4) Universal Description, Discovery and Integration (UDDI) is used as the directory service. In addition, although not part of the basic implementation, there are over 100 standards to support other system qualities such as WS-Security for security and WS-ReliableMessaging for reliability.

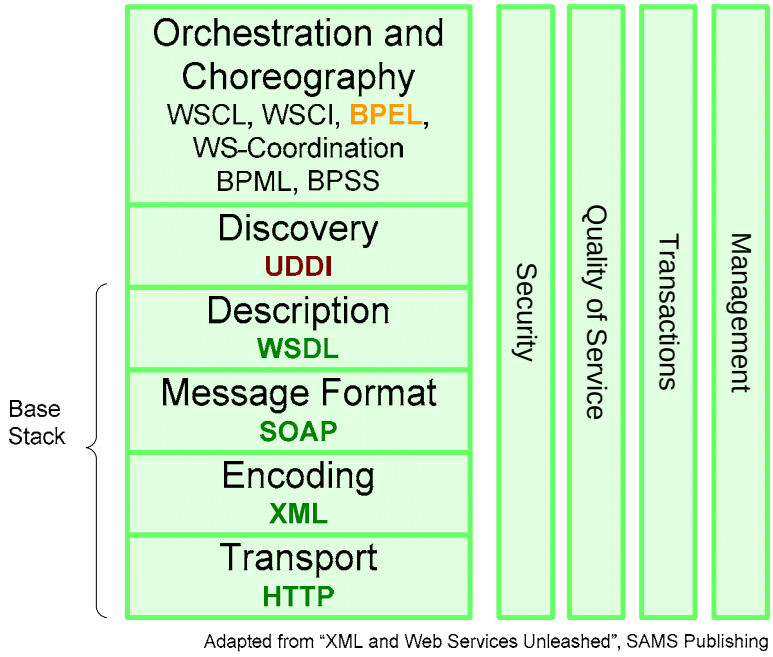

Figure 3: WS Web Services Protocol and Standards Stack*

The following table summarizes how standardization—in aspects such as service description, service discovery, message formats, encoding, and transport—impacts quality attributes in a service-oriented system.

Table 1: Effect of Standardization

Quality Attribute	Effect	Explanation
Interoperability	Positive	Standardization in service-oriented system implementations is the primary enabler of interoperability, both across platforms and vendors. However, policies and standards have to be in place to increase interoperability through consistency (e.g., service interfaces, data models, implementation technologies, and versions of infrastructure elements).
Modifiability	Positive	• Standardization at the service interface level enables service providers to change service implementations (e.g., to port a service implementation to a new language) without a major effect on service consumers, as long as the service interface and the expected behavior of the implementation remain unchanged. • Standardization at the infrastructure level, mainly for integration between components, enables infrastructure components[5] to change as technology changes without as many effects on consumers. • The use of XML as a form of standardization provides flexibility in contract specification to deal with changes without having major effects on service consumers (e.g., new elements in an XML schema can be made temporarily optional, XML schemas can have extension points if certain changes are anticipated).

[5] Infrastructure components are elements of the SOA infrastructure shown in Figure 2. Examples of an infrastructure component include ESB, service registry, load balancer, and monitoring tools.

Quality Attribute	Effect	Explanation
Performance	Negative	• The use of XML for standardization in most service-oriented implementations has a negative effect on performance because XML involves three CPU-intensive activities: parsing, validation, and transformation [Juric 2004]. • Some standards (e.g., SOAP) require bridges, proxies, stubs, and similar elements to perform transformations and translations that are needed for interoperability.
Reusability	Positive	• Standardized service interfaces enable services to be used across multiple business and operational processes. However, proper service identification processes have to be in place such that all services represent independent and cohesive business/operational tasks. • Standards are often used to promote reuse between service providers and consumers that use heterogeneous technologies. However, the parsing of these standard formats incurs significant runtime overhead which contributes to latency of single operations.
Security	It depends	• In the case of WS* Web Services, these inherit all the attacks from all the standards used in their implementation, such as XML Denial of Service (XDoS) attacks [Sullivan 2009]. • From a security standards perspective, WS-Security is the emerging standard for web service security and accommodates multiple security models and encryption technologies [OASIS 2006]. However, WS-Security by itself does not define a security solution; it simply provides the elements to create a security solution. For example, there must be an agreement between service consumers and providers on certain elements such as the security tokens to be used for authentication [OASIS 2006]. All these agreements, as well as the tools to generate and validate tokens, are part of the security solution.
Testability	Positive	Standardization enables the use of commercial testing tools for service-oriented environments as well as additional forms of testing that take into account that system components might not be available at design time, such as interface-based testing [Ghosh 2000].

3.2 Loose Coupling

Loose coupling is one of the key goals for SOA adoption. There are multiple areas where an architect can make decisions to promote loose coupling. If loose coupling is an important architectural driver, an architect should:

- Ensure that service providers and service consumers are independently able to make decisions about technology. A service consumer should not be bound to a particular technology (e.g., programming language, database, platform) in order to use a service. Service consumers should be allowed to select technologies that best fit their organizational context and policies, enterprise architecture, legacy systems, licensing model, and development model. The architect should define how messages will be exchanged, how messages will be formatted, and how services will be described. SOA implementations that are built using web services use XML for message exchange and service description and stable standards such as SOAP and HTTP for communications. These constraints make loose coupling possible in service-oriented environments. Distributed computing technologies such as CORBA and COM/DCOM result in tighter coupling between the consumer and the provider because they have to share the same interface (with no intermediary), the consumer has to install and run specific components, and the consumer has to create an execution context that is maintained during the interaction.

- Create services that are independent, self-contained capabilities that can be used in isolation from capabilities provided by other services. Services should react to messages by performing the required operations based on message content. The service should have no knowledge of how the response will be used, the order in which it needs to be invoked, or how the input message was created.
- Enable the actual binding between a service consumer and a service at runtime. The later you defer binding the more flexibility service providers and service consumers have to develop their software systems independently. In an ideal situation, the only agreement required is the service interface description, which includes a set of ordered messages that have a defined schema. Therefore, assuming that the service interface remains unchanged, the service consumer can elect to use an alternative service that abides by the agreement without code changes and the service provider can change service implementations without affecting the service consumer.

The following table summarizes how achieving the goal of loose coupling impacts quality attributes in a service-oriented system.

Table 2: Effect of Loose Coupling

Quality Attribute	Effect	Explanation
Interoperability	Positive	Standardization of data representation and service description inherent to service-orientation enables loose coupling of technologies chosen by different organizations that need to interoperate.
Modifiability	Positive	Loose coupling between service provider and service consumer enables each to change implementations independently as long as the service interface and the expected behavior of the implementation remain unchanged.
Performance	Negative	• The use of standards to promote loose coupling can have a negative effect on performance when CPU- and memory-intensive activities, such as parsing, validation, and transformation are required [Juric 2004]. • Run-time binding is more expensive than static binding, but the impact can be limited to a single one-time cost.
Reliability	It Depends	Loose coupling enables service consumers to move to alternative service implementations in case of service failure; redundancy can be implemented in the case of stateless services. Self-contained services can also promote reliability by constraining the propagation of failures. However, the potential lack of control over service elements (e.g., third-party service implementations) introduces unpredictability.
Reusability	Positive	Because services are not bound to a particular implementation or technology, it is easier to reuse them. In addition, because of the loose coupling between services, it is easier to reuse services without worrying about the dependencies between them. However, proper service identification processes have to be in place such that all services represent independent and cohesive business/operational tasks.
Scalability	Positive	Because of loose coupling it is possible to create a scalable architecture in which new service instances are added on-demand to meet increased loads.
Security	Negative	• Overall system security can be affected because of lack of control over system elements. Services developed by different organizations may not have practices to ensure that service implementation code is written securely. • The use of XML to enable loose coupling introduces new security threats such as XML Denial of Service (XDoS) [Sullivan 2009].

3.3 Reusability

The goal of increasing reuse in this context is mostly associated with service reusability. Services are reusable because they represent self-contained functionality that can be used in multiple business processes. If reusability is a business goal, an architect may employ the following strategies:

- Identify services that perform utility operations (i.e., logging) and data manipulation and design them so that they are independent of the business processes that use them. This is known as abstracting common services to promote reusability. This will avoid unnecessary duplication of logic that performs these operations as part of multiple business processes.
- Design services to be stateless when possible. Services should not maintain conversational state. Each service request is self-contained and independent of other requests. Even though it is possible to build stateful services, there are advantages of stateless services for reusability. Stateless services can be replicated more easily to promote many different QoS requirements such as availability (redundancy) and scalability to meet additional demand (load balancing).
- Provide infrastructure that abstracts or mediates differences between service consumers and service interfaces such as binding technologies or standards. An example is an Enterprise Service Bus that implements the VETRO (Validate – Enrich – Transform – Route – Operate) pattern to deal with differences between service consumers and providers [Chappell 2004].
- Provide mechanisms to allow consumers to find a service that meets their needs. This will be discussed in Section 3.5.

Promoting reusability has an impact on other quality attributes. The table below provides a general assessment of how quality attributes are impacted.

Table 3: Effect of Reusability

Quality Attribute	Effect	Explanation
Interoperability	Positive	Service reusability also depends on providing standard interfaces to reusable service functionality, therefore increasing interoperability between service consumers and services.
Maintainability/ Evolution	Positive	Reusable services and other assets provide common functionality that reduces the number of instances of logic that need to be maintained. When changes are required there is usually less effort required.
Performance	Negative	Techniques for promoting reusability are often at conflict with performance. Standards are often used to promote reuse between service providers and consumers that use heterogeneous technologies. The parsing of these standard formats incurs runtime overhead, which contributes to latency of single operations.
Reliability	It Depends	Increased reusability means that a potentially greater number of service consumers will be affected if there are problems with a service or other reusable assets. As the number of service consumers that depend on a reusable service grows, the more important it is that the architecture has reliability mechanisms to detect and recover from failures.
Scalability	It Depends	It is possible to identify services with high usage and create a scalable architecture to deal with services that have high demand. In the case of stateless services, introducing redundancy is simplified. Stateful services add an additional requirement for synchronization between two instances of the same service if they are replicated, which introduces complexity.

Testability	Positive	Reusable services and other assets that have been unit tested can be reused without having to be retested.

3.4 Composability

The end goal of composability is to be able to change pieces of a business process rapidly when the business environment changes, without impacting the consumers of the composite service that implements the business process.

Service composability depends on many of the same service characteristics as service reusability: self-contained functionality, standardized interfaces, and availability in a service registry. It also relies on proper service identification and careful service interface design. The difference between composability and reusability is that composability relies on a set of reusable services that have properties that enable them to be composed. A composition may rely heavily on infrastructure such as an orchestration engine for the choreography of services based on business workflows.

Composability introduces architectural risks that require strategies for mitigation. Below are some examples:

- In self-contained application transactions it is relatively simple to commit changes to system state for successful operations or to roll back to a previous state when failures occur. In a service composition there are often partial failures that introduce inconsistent states that need to be repaired. This is complicated because the underlying services are often implemented using different technologies. Web services provide standards for distributed transactions, but there are many interoperability issues between the vendor products that are part of the solution. A possible mitigation strategy is to commit to a single vendor platform but this creates conflicts with other drivers for SOA adoption (e.g., reusability, interoperability). Other mitigation strategies include
  - Retrying failed operations a specified number of times. This requires all service operations to be idempotent (i.e., duplicate requests are not processed).
  - Creating service agents that have logic to perform inverse or compensating operations. This is possible but often extremely complex.
- Creating compositions with services that span multiple organizational boundaries can introduce significant architectural risks because of the loss of control. The QoS of each service can change radically over time. A simple change in deployment (i.e., installing a patch to remove a vulnerability) can have major effects on latency. Possible mitigation strategies include
  - Negotiate service level agreements that specify QoS and the penalties for violations of that agreement. It is important to note that service level agreements often do not provide adequate compensation for actual damages.
  - Select providers that collect metrics and provide specialized interfaces to access current data relating to important QoS requirements to determine if they meet the response measures (e.g., average latency, service availability) of your architectural drivers.
  - Conduct experiments using a set of synthetic transactions and collect metrics associated with the response measures of your architectural drivers.

The following table summarizes how composability impacts quality attributes in a service-oriented system.

Table 4: Effect of Composability

Quality Attribute	Effect	Explanation
Interoperability	It Depends	• Tools that support service composition are likely to have constructs and technologies to deal with mismatches between participating services. • Standards for composing services introduce cross-vendor interoperability challenges.
Performance	Negative	• Orchestration engines, if used instead of point-to-point compositions, introduce an additional layer of computation. • Performance is determined by the lowest performing service in the composition. If composed services are executed serially rather than in parallel, performance is determined by the sum of each service performance.
Reliability/ Availability	Negative	• Service compositions may require transactional behavior that is complex to manage in a service-oriented environment because participating services are potentially distributed, multi-platform, and even multi-organizational. • The availability of the composition is calculated by multiplying the availability of each member (e.g., .99 * .99 *.99, produces an availability of approximately 97%).
Security	Negative	Increased composability means that services have the potential to be composed in ways that original designers never envisioned. This may cause inadvertent disclosure of sensitive information through aggregation. The classic example is when two or more pieces of "benign" data are fused together and become proprietary or classified information.
Testability	It Depends	• Service compositions that have already been tested can be reused without having to conduct unit and integration tests on the composition. Unfortunately, the compositions may require retesting when used in different systems depending on risk tolerance. • Changes to individual participating services trigger retesting of all the compositions that the service participates in.

3.5 Discoverability

In a service-oriented environment, services are created and published in a place that is accessible to service consumers (e.g., service registry, web page, directory, etc.). Ideally, service consumers can query this service registry looking for services that satisfy desired capabilities. At a minimum, the metadata associated with a service is its interface specification or contract. Additional metadata associated with a service is commonly stored in a service repository[6] and includes attributes such as

- Description
- Classification
- Usage history
- Test cases
- Test results

[6] Even though service registry and service repository are often used interchangeably, we use service registry to describe a system element similar to a searchable directory and service repository to describe a system element that stores additional metadata and artifacts associated with services registered in the service registry.

- Quality metrics
- Documentation
- Sample code

Service discovery in practice is often done at design time. The developer of the service consumer queries the service registry at design time and obtains the necessary information in order to invoke the service. Discovery depends on the availability of the service registry as well as the quality of the information in the registry. High availability of the registry at design time is unlikely to be required and the quality of the information is a governance issue, which we will not address in this report. We will focus on the dynamic aspects of service discovery.

Dynamic service discovery refers to service discovery that happens at runtime. Unfortunately, the word *dynamic* is used in many ways to describe the binding between service consumers and services. There are various degrees of dynamism. The architect needs to decide the appropriate level of dynamism required to meet architecturally significant requirements. Below are some decisions relating to varying degrees of dynamism:

- At the lower end of the spectrum an architect can choose to create proxy services that serve as gateways for binding to specific service instances based on runtime conditions or user properties. These proxies can provide many benefits such as location independence, request filtering, and load balancing. This level of runtime binding is a common, out-of-the-box feature of many commercial and open-source SOA infrastructures, such as an enterprise service bus (ESB). A registry would not be required.
- At the higher end of the spectrum is fully dynamic binding in which service consumers are capable of querying service registries at runtime, selecting the "best" service from the list of returned services, and invoking the selected service—all at runtime, and without human intervention. There is an important tradeoff with fully dynamic binding. It requires semantically described services that use an ontology or data dictionary that is shared between service consumers and service providers such that there is a common understanding of the meaning of exchanged data and the relationships between data elements. Semantic Web Services represent an active area of research, as well as an unsolved problem that is not yet ready for large-scale deployment.

The following table summarizes how dynamic discovery impacts quality attributes in a service-oriented system.

Table 5: Effect of Discoverability

Quality Attribute	Effect	Explanation
Interoperability	Positive	Because of late or runtime time binding, the infrastructure can be set up to check the registry at runtime and point to different versions of a service depending on service consumer or message characteristics.
Maintainability/ Evolution	Positive	Service registries may be set up to have pointers to different versions of services such that changes in services have minimal impact on existing service consumers.

Quality Attribute	Effect	Explanation
Performance	It Depends	It is common for service registries to be used by service consumer developers at design time, which means that the registry would not have an impact at runtime. However, if the infrastructure is set up such that the registry is checked at runtime or the registry performs some form of selection between different versions of a service, there is an additional computation involved that affects performance. These effects can be reduced by reducing the number of times the registries need to be queried.
Reliability	It Depends	Because of late binding, the infrastructure can be set up to check the registry at runtime and point to alternate instances of services depending on error conditions such as service unavailability.
Reusability	Positive	An important benefit of a service registry that supports service discovery is reusability of services. The registry can be used so that designers can find services that fit their needs to avoid unnecessary duplication of functionality.
Scalability	It Depends	Because of late binding, the infrastructure can be set up such that all compatible instances of a specific service are checked at runtime for current utilization and the request is forwarded to the service that has the lowest utilization. However, if the registry has to be checked at runtime, it may become a bottleneck.

4 Common Components of a Service-Oriented System

As shown earlier in Figure 2, the main elements of a service-oriented system are service consumers, services (interface plus implementation), and the SOA infrastructure. The SOA infrastructure plays an important role in service-oriented systems because it mediates differences between service consumers and providers therefore promoting important quality attributes such as interoperability, modifiability, and extensibility. What follows are some of the components that are commonly part of a SOA infrastructure, its supporting patterns and tactics and the impact that these components have on overall system quality.

4.1 Enterprise Service Bus

An Enterprise Service Bus (ESB) is a software pattern that can be part of a SOA infrastructure and acts as an intermediary between service consumers and service providers. Service consumers are designed to interact with the ESB and the ESB is configured to route and transform different kinds of request and response messages between service consumers and service providers. There are vendor products that implement many of the features described below in the supporting patterns and tactics section. It is important to understand that an ESB is not required in order to implement a service-oriented system. In certain contexts point-to-point integration between service consumers and providers makes sense. In homogeneous environments that are under a single organization's control, an ESB may be overkill.

4.1.1 Supporting Patterns and Tactics

ESB is a "compound" pattern that consists of the following patterns, as shown in Figure 4 [Erl 2009]:

- **Intermediate Routing**: A generic router service intercepts messages (service requests and responses) and based on routing logic determines where the messages should be forwarded. The routing logic may be augmented to support many tactics:
  - Load balancing (e.g., fail-over to a backup in case the primary destination service is unavailable)
  - Service version selection (e.g., requests are sent to compatible versions of a service in order to support backwards compatibility during transition periods)
  - Service selection based on message data (e.g., requests from premium clients are sent to a faster processing component)
  - Access control rules (e.g., a request from an unauthenticated user is routed to a login page)
  - Exception handling (e.g., a response error message that is rerouted to a service responsible for centralized exception handling)

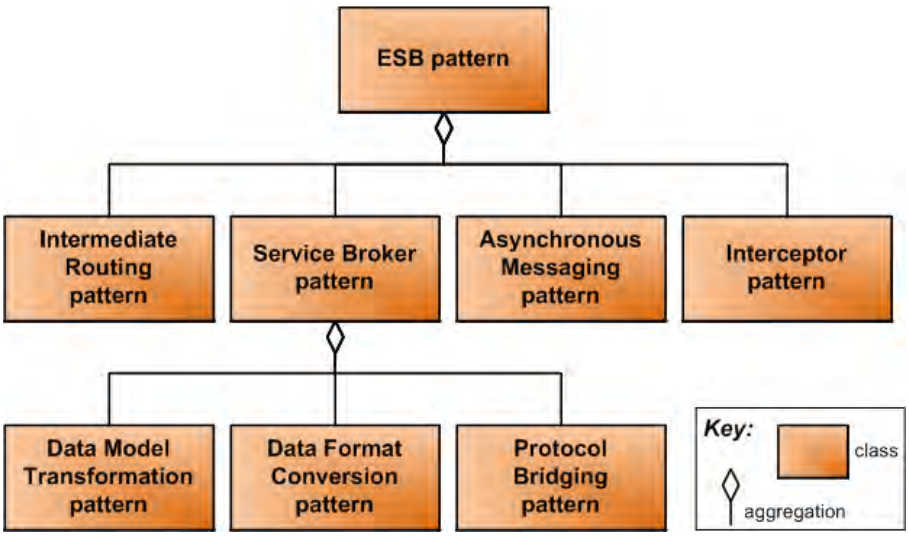

Figure 4: ESB Patterns and Sub-Patterns (adapted from [Erl 2009])

- **Service Broker**: SOA solutions often integrate components that were developed at different times or by different organizations. It is not unusual to find mismatches between the required interface of a service consumer and the provided interface of the respective service provider. These mismatches may result from differences in (1) the structure of the data exchanged, (2) the data format representation used for the message content and (3) the communication protocol or component model. The service broker pattern is itself a compound pattern that prescribes the use of an intermediary (the ESB) that deals with these mismatches by applying the following three other patterns:
  - **Data Model Transformation**: Data sent from the service consumer in a given structure is transformed into a different structure that is expected by the service provider (and likewise for the data sent back as a response from the provider to the consumer). In XML terms, for example, this pattern has the service broker applying an XSLT transformation from one XML schema to another.
  - **Data Format Conversion**: Used whenever service consumer and provider need to exchange data represented in different formats (e.g., XML, CSV, JSON).
  - **Protocol Bridging**: The service consumer sends a request using a protocol (e.g., SOAP version 1.2 over HTTP) and the service broker intercepts the request and converts it to a request to the service provider using a different protocol (e.g., Java Remote Method Invocation (RMI)). The protocol mismatch between service provider and consumer can vary from a simple version mismatch for the same protocol (e.g., SOAP v1.1 versus v1.2) to the use of different protocols in one or more protocol layers (e.g., SOAP over HTTP versus SOAP over JMS (Java Messaging Service)).
- **Asynchronous Messaging:** Some ESB products provide messaging system capability and allow service requests and responses to be exchanged via messaging channels. The messaging channels can be configured as point-to-point queues or publish-subscribe topics. Queuing inside the ESB may be transparent to participating service consumers and providers. Although messaging channels are often used for asynchronous messaging, ESB products can also use these channels for synchronous business transactions using request and response queues and callback addresses in the service configuration.

- **Interceptor**: Some ESBs offer the ability to configure interceptors, which are software elements that are activated for all requests and responses. An interceptor is called by the ESB and receives the service request (or response on the way back) as a parameter. The interceptor can be used for cross-cutting functionality such as logging, authorization checks, and performance profiling. After successful execution, the interceptor lets the message proceed to its destination or the next interceptor in the chain.

4.1.2 Impact on System Quality

Table 6 and Table 7 explain respectively the negative and positive effect on different quality attributes related to the use of an ESB in a SOA solution. Not all ESB products provide all the features discussed in this report. The quality impact considerations discussed below only apply to products that provide the corresponding feature.

Table 6: ESB Aspects that Negatively Affect System Qualities

Quality Attribute	Explanation of Negative Effect
Modifiability	If the ESB performs intricate data model transformations, a substantial part of the service consumer and provider interaction logic becomes codified by the transformation rules (e.g., XSLT stylesheets). These transformation rules add to the complexity of developing and maintaining the solution.
Performance[7]	• The ESB is an intermediary and hence produces a communication overhead if compared to direct point-to-point communication between service consumer and provider. The processing done by the ESB (e.g., routing logic, data model transformation, data format conversion, protocol conversion) adds to the roundtrip time of the service execution. • The routing logic can often be changed dynamically. The ability to configure routing rules at runtime or even at load time typically incurs a performance overhead related to loading routing data from configuration files and interpreting routing rules. • If the ESB is providing reliable messaging, there is a performance overhead involved with message persistence and acknowledgement notification.
Security	• The ESB is another component to protect and a complex one. Misconfigured or corrupted routing logic may result in unauthorized access.
Availability	• The ESB may be a single point of failure in the system.

[7] It is important to note that performance is not always about overhead. Some additional overhead is acceptable if predictable performance is attained. There are many fine-grained decisions that affect predictability such as service design, message size, operating systems, and protocols.

Table 7: ESB Aspects that Positively Affect Systems Qualities

Quality Attribute	Explanation of Positive Effect
Interoperability	The ESB allows disparate systems to interoperate in spite of mismatches in data models, data representation formats, communication protocols and implementation technologies. This capability is particularly important to integrate legacy systems and silo applications that run on different platforms.
Modifiability	• The ability to perform data model transformation enables the deployment of new versions of a service without disrupting existing service consumers. Service requests using data models of old versions are transformed to adhere to the current version of the service interface. • Data format conversion and protocol bridging capability increase the ability of the system to easily incorporate new service consumers or providers that use different data formats and technologies. Because format and protocol conversion are typically provided out-of-the-box in ESB products, they do not incur an implementation complexity penalty.
Reliability	When the receiver of a service request or response has failed, the ESB may queue the message until the service is available again. The internal queue of requests and responses is sometimes persistent, which yields even better reliability.
Security	The ESB may include access control functionality. It may enforce authentication and authorization rules in service message exchanges.

4.2 Service Registry and Repository

Service registries and repositories can be custom built, but are often provided by a product in the SOA infrastructure. Vendor products support a subset of the functionality listed below:

- **Dependency Management:** Provide automatic dependency detection and the ability to specify certain dependencies (e.g., use of another service for input validation) to aid architects when performing change impact analysis. Even though this feature is commonly used for build automation, it can also help architects in root cause analysis of service failures.
- **Discovery:** Support the ability of consumers to query the registry to find services that fit their needs. In addition to desired capabilities, query criteria can include supported standards, security policies, and QoS parameters specified in a service-level agreement (SLA).
- **Versioning:** Offer multiple versions of the same service, each version possibly with a different interface and SLA.
- **Event Notification:** Allow stakeholders (e.g., developers, consumers, business process owners) to subscribe to notifications of changes to registry content of interest (e.g., service interface changes).
- **Access Control:** Provide security mechanisms to protect unauthorized access to service artifacts such as WSDL descriptions, BPEL descriptions, and policies.
- **Policy Management:** Service registries support the specification of policy assertions that a service consumer must meet prior to service invocation. The classic example of a security policy assertion is that all inbound messages must use the SHA-1 hashing algorithm for message integrity and 256-bit encryption.
- **Federation Capabilities:** The ability to integrate multiple registries so that they appear to be a single registry from the service consumer perspective.

4.2.1 Supporting Patterns and Tactics

Some supporting patterns and tactics for service-oriented systems include:

- **Canonical Expression:** Service contracts are standardized with naming conventions to avoid inconsistency that could lead to a service catalog that is ambiguous. The goal is to make service contracts consistently understood and interpreted [Erl 2009]. An example of this pattern would be consistent naming of service operations across enterprise-wide services. This pattern supports service-oriented principles of reusability and discoverability.
- **Metadata Centralization:** As organizations increase in size, the risk of producing functionality that already exists also increases. This can result in wasted effort and can create inconsistencies in how common operations are implemented. To prevent these issues, service metadata is published in a central service registry to enable discovery. This metadata is usually controlled by formal processes for publication so that resources that make it into the registry are largely enterprise-wide services that are agnostic to functional context [Erl 2009].
- **Canonical Versioning:** A service inventory that has different policies for version control has the potential to create modifiability and interoperability challenges. These challenges can negatively impact service development and reuse as well as dynamic access to services. This pattern often leverages existing versioning and configuration management policies in an organization [Erl 2009]. A typical example of a service versioning policy is how long an existing service contract will be supported after a new version of the service contract is released.
- **Publish-Subscribe:** Service registries often allow service consumers to register to receive notification of service contract changes. It can often also be set up such that change notifications are automatically sent to all consumers.
- **Use of an Intermediary:** A service registry provides location transparency such that the physical location of a service can be changed without requiring modification to service consumers.
- **Maintaining Existing Interfaces:** Service registries support multiple versions of services to maintain backward compatibility for periods of time when service contracts are broken to allow service consumers to continue to use a service until the required changes can be made on the service consumer side.
- **Adherence to Defined Protocols:** Service registries support standards for description, policy assertions, and data schemas.
- **Runtime Registration:** Services can be registered and added to a service registry at runtime.
- **Multiple Registry Copies:** Many registry products support clustering multiple instances to insure that registry/repository is not a single point of failure. This tactic also can improve registry throughput through load balancing of requests.

4.2.2 Impact on System Quality

Table 8 and Table 9 explain respectively the negative and positive effect on different quality attributes related to the use of a service registry and repository in a SOA solution.

Table 8: Service Registry and Repository Aspects that Negatively Affect System Qualities

Quality Attribute	Explanation of Negative Effect
Availability	A registry that is not replicated can be a single point of failure.
Performance	Using the registry for dynamic discovery and binding of services and consumers increases latency.
Security	Exposing details of service metadata can provide useful information for attackers who can compromise services.

Table 9: Service Registry and Repository Aspects that Positively Affect System Qualities

Quality Attribute	Explanation of Positive Effect
Availability	Service registries can be used at runtime (i.e., procedures to retry requests) to determine if a suitable replacement for a service that has failed can be found and invoked.
Interoperability	• Multiple versions of the same service that use different approaches for communication (e.g., SOAP, REST) or different data formats (e.g., XML, JSON) can be made available in the service registry. Service consumers can find compatible services by querying the registry. • The standards for service description and policy assertions are mature.
Modifiability	• The service provider and consumer are loosely coupled and binding can be delayed until runtime. • Change impact analysis can be informed by the dependency management capabilities found in many service registries. • Multiple versions of a service can be maintained to allow time for service consumers to migrate to use new versions of a service.
Security	Consumers can query the service registry for services that provide appropriate message-level security, confidentiality, etc.

4.3 Messaging System

A messaging system, also known as message-oriented middleware, is often part of the execution environment of enterprise applications. It allows distributed components to exchange asynchronous messages. Messaging systems have existed long before the advent of SOAP and other Web Services standards. Today they are commonly used in SOA solutions for transactions that involve background processing because they can provide high levels of scalability and reliability. In fact, messaging system capability has been embedded or integrated in many ESB products and business process engines.

4.3.1 Supporting Patterns and Tactics

In a service-oriented system, service consumers and providers can communicate via asynchronous message exchanges, which are routed via the underlying infrastructure (messaging system). The basic asynchronous messaging pattern is complemented and specialized by messaging-related patterns as shown in Figure 5. These messaging patterns applicable to service-oriented systems are just a subset of a vast collection of known enterprise integration patterns [Hohpe 2003]. Some other messaging patterns can include:

- **Service Callback:** This pattern is used when a service consumer sends an asynchronous message to a service provider and the service business logic requires the service provider to

send back one or multiple response messages to the service consumer. In this case, the original request contains a callback address to which the service provider sends the response messages. The callback address can be a port of the service consumer or can point to a completely different location. Service callback makes asynchronous messaging an alternative to RPC-like interactions (e.g., as in SOAP-based web services) and has the benefit that the caller is not blocked while waiting for a response. This pattern is particularly useful for services that take a long time to process or long-running business processes in which the consumer needs to keep track of step completion.

- **Correlation Identifier**: When messaging is used to asynchronously send a request in a request-reply interaction, the sender needs a way to correlate the original request to the reply message (the callback) that also arrives asynchronously some time later. One solution is for the sender to make only one request at a time so that there is only one outstanding request at any time. But this alternative creates throughput and scalability issues and is often not acceptable. A better alternative is to extend the message content[8] to include a correlation identifier—a unique identifier that is added to the request and later added to the callback so that the sender can correlate both messages.
- **Reliable Messaging**: The goal of this pattern is to guarantee message delivery in case of failure. The queues are persisted so that messages in the queues are not lost in case the messaging system crashes. Once a message is successfully delivered it can be removed from the temporary storage. Reliable messaging may also include positive and negative acknowledgements for each message or group of messages [Erl 2009].
- **Point-to-Point Channel:** Service providers and consumers communicate through a message channel (queue), which is a unidirectional conduit of messages. The service consumer sends a service request message to the queue and does not get blocked while waiting for a response—unlike synchronous RPC-like interactions. There might be different components and multiple component instances that get messages from the same queue, but each message is delivered to only one receiver. Depending on the messaging system features and the configuration of the messaging channel: (a) messages may be delivered in the order of arrival or according to the message priority, (b) the sender may or may not receive an acknowledgement that the message was delivered and (c) the point-to-point channel can follow the push or the pull model:
  - Push: The messaging system signals the message receiver component of the arrival of a message. The receiver needs to be pre-registered with the queue.
  - Pull: Message receivers need to poll the queue periodically to retrieve messages.
- **Publish-Subscribe Channel:** In this case, the messaging system allows components to subscribe to a message channel (often called "topic"). The message sent to the topic is forwarded asynchronously to all subscribers of that topic. Subscribing and unsubscribing (or closing the connection to the message channel) can be done at runtime. Each subscriber receives the same message no more than once. The message is considered consumed when all

[8] The correlation identifier is not part of the service input or output data; it is added to the header (as message metadata) rather than to the body of the message.

subscribers have received it or the message expires because of timeout. Some messaging systems have the option of a durable subscriber—when the subscriber is not available to receive a message the system saves the message for a certain amount of time. When the subscriber reconnects, it receives all messages that were saved by the messaging system. Similar to a point-to-point channel, a publish-subscribe channel may use the push or the pull model.

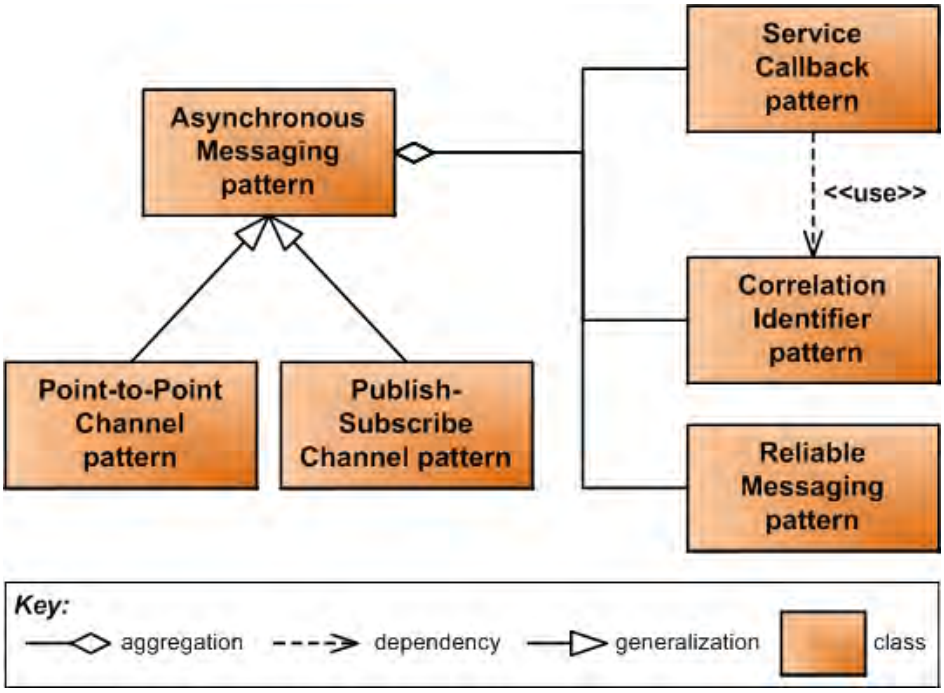

Figure 5: Asynchronous Messaging Pattern, Specializations, and Sub-Patterns

4.3.2 Impact on System Quality

Table 10 and Table 11 discuss how different quality attributes of a SOA solution are negatively and positively affected by the use of a messaging system.

Table 10: Messaging System Aspects that Negatively Affect System Qualities

Quality Attribute	Explanation of Negative Effect
Interoperability	The use of the same API for implementing the message producing and message consuming code (e.g., JMS API in the Java world) does not guarantee interoperability. The message producer and the message consumer services have to use locally the same messaging system product or compatible products. Most messaging systems products use proprietary wire protocols for communication. The alternative is to introduce a bridge connector. For example, a service running on a platform that contains Microsoft MQ can send a message to an IBM MQSeries destination queue on a different computer through an MSMQ-MQSeries bridge. Some ESB products provide this bridging capability.
Modifiability	• The asynchronous message paradigm is more complex because it requires a callback. The sender of the original message has to implement a callback endpoint and deal with the waiting and correlation of responses. • When an asynchronous message is sent, both service consumer (message sender) and provider (message receiver) execute in parallel. The concurrent execution of different parts of the same transaction requires extra caution when accessing resources, and adds complexity to the exception handling logic (e.g., retries, compensating operations).

Performance	Going through the messaging system incurs an overhead in the overall processing due to queue processing delays and persisting messages in the queue. Thus, messaging systems are more advisable when there are no stringent timing requirements for message processing.
Reliability	Atomic transactions that embody tasks activated via asynchronous messages are often infeasible.

Table 11: Messaging System Aspects that Positively Affect System Qualities

Quality Attribute	Explanation of Positive Effect
Interoperability	• There are emerging standards for the wire-level protocol used by messaging systems that should allow different systems to interoperate. One of them is advanced queueing message protocol (AMQP), an open standard initially proposed by J.P. Morgan and now under the responsibility of the AMQP Working Group [AMQP 2010]. Another standard is Stomp, which is a text-based protocol specification published as part of an open source project hosted by codehaus.org [Stomp 2010]. • The WS-ReliableMessaging standard allows message producers and consumers implemented in different languages and on different platforms to interoperate using the SOAP protocol [OASIS 2007]. Support for this emerging standard has been announced by a number of middleware vendors.
Modifiability	If the service consumer and provider can communicate via messages it is easier to insert a mediator of that interaction.
Performance	• Service requests are processed by message consumers in the background with no blocking time for the service consumer. • Publish-subscribe channels are efficient in local networks because the messaging system can take advantage of the Ethernet bus architecture and use IP Multicast [Hohpe 2003]. In this case, a single IP packet is sent and multiple recipients can read it.
Reliability	Many messaging systems offer reliable messaging with persistent queues and guaranteed delivery of messages.
Scalability and Availability	• Horizontal scalability in some messaging systems can be achieved by creating clusters of replicated messaging system nodes. These cluster configurations provide load balancing, high availability, and automatic failover.

4.4 Business Process Engine

A business process engine is a software component responsible for processing incoming requests by performing the steps of the business process that correspond to that request. These steps typically involve calling one or more services. Part of the business process engine solution is a business process modeling (BPM) tool that allows the description, and often visualization, of a business process. The BPM tool then generates a business process[9] that is deployed to and executed by the business process engine when processing requests.

The services that participate in the execution of the business process do not interact directly with each other; they interact with the business process engine that coordinates the execution. Because of this coordination role, the business process engine is often called an orchestration engine or orchestration server. A business process engine may or may not be part of the SOA infrastructure, but if it is not, the business process has to be coded manually as part of the service implementation or in the service consumer.

[9] The BPM tool stores the business process specification in a given language (for example, web services business process execution language (WS-BPEL)). For deployment to the process engine, the business process may be compiled into a different language used by the business process engine.

4.4.1 Supporting Patterns and Tactics

The business process engine follows the SOA design pattern called orchestration [Erl 2009]. This pattern is a compound pattern—it results from the combination of four other patterns, as shown in Figure 6.

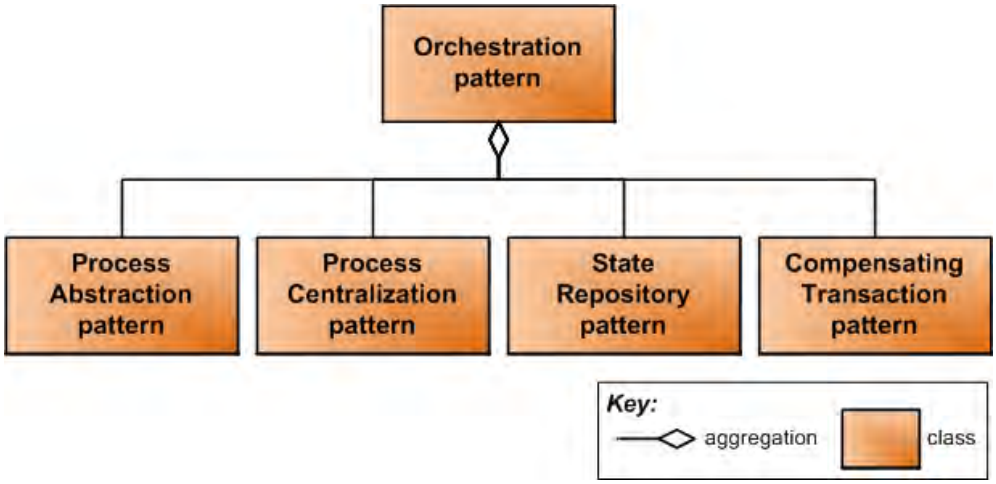

Figure 6: Orchestration Pattern and Sub-Patterns (adapted from [Erl 2009])

The four patterns that make up the compound orchestration pattern are:

- **Process Abstraction**: This pattern prescribes that the business logic specific to a given business process should be implemented in a task-based service, basically following the layered structure described in Section 2.3. The task-based service does not contain logic that is reusable across services, such as data access and utilities. Instead, the task-based service is composed by calling entity and utility services. This separation of services in layers enables reuse and service composition, which are important to successfully employ a business process engine solution.
- **Process Centralization:** The Process Abstraction pattern provides a business logic layer. However, the task-based services in that layer may be implemented independently and scattered across the organization. The Process Centralization pattern recommends that these services be placed in a centralized location. Furthermore, an orchestration engine becomes responsible for executing the business logic services. Instead of being manually coded, these services can be defined using a BPM tool and compiled to a format readable by the orchestration engine. The orchestration engine mediates the interaction with entities, utilities, and other services that participate in the business process, and also provides special capabilities, such as fault handling, life-cycle management for long-running asynchronous service interactions, human tasks as part of the workflow, notifications, and parallel execution of services.
- **State Repository**: The execution of a business process may involve the invocation of multiple services. Data returned from a given service invocation may not be used until much later in the process workflow. To avoid holding the data in memory for extended periods, this pattern introduces a repository for temporarily storing the business process state. Despite the performance overhead to persist data and retrieve it later when it is needed, the state reposi-

tory decreases overall memory consumption and improves reliability of the system because state may be restored from the repository upon a failure. Temporarily storing process state in a database is sometimes called dehydration. Dehydration points may be designed into the business process, but can also be inserted automatically by some business process engines.

- **Compensating Transaction**: The execution of a business process may fail because an exception occurred in a participating service or in the business process engine. In this case, database changes made by the participating services may need to be rolled back to the previous state. Often these services do not run in the context of a single database transaction or a distributed transaction. The Compensating Transaction pattern provides an alternative that consists of implementing service compensating logic in each service. This logic is invoked by the business process upon a failure and performs an "undo" of the changes previously made by the service. For each operation in the service interface that alters persistent data, an undo operation is added to the interface.

In addition to service orchestration, some business process engine products have capabilities that promote interoperability. These products implement the following patterns, which were discussed in Section 4.1.1:

- Data model transformation
- Data format conversion
- Protocol bridging

4.4.2 Impact on System Quality

Table 12 and Table 13 discuss the negative and positive effects in different quality attributes when a business process engine is used in a SOA solution.

Table 12: Business Process Engine Aspects that Negatively Affect System Qualities

Quality Attribute	Explanation of Negative Effect
Modifiability	• The need to design and implement compensating transactions into services increases the complexity of the solution and can increase the size and complexity of service interfaces. • The ability to dynamically modify routing rules that are part of a business process workflow has the risk of having untested logic running in production. Thus, such changes should go through traditional testing phases.
Performance	Upon the specific event that triggers a business process, the business process engine executes the business process and intermediates the interaction of the participating services. That orchestration role incurs a performance overhead.
Security	• The business process engine is another component to protect. Misconfigured or corrupted workflow logic may result in a security breach. • The business process engine may be a single point of failure in the SOA infrastructure (a security and availability concern).

Table 13: Business Process Engine Aspects that Positively Affect System Qualities

Quality Attribute	Explanation of Positive Effect
Interoperability	Some business process engines are capable of interacting with services through different protocols (e.g., SOAP over HTTP, SMTP, JMS) and therefore promote seamless integration of these disparate services.
Modifiability	• Externalizing business process flow logic from a service's code allows easier implementation of business rules. Business process workflows can more easily be changed if a visual workflow modeling tool is available. • Entity- and Utility-based services are the best candidates for reusable logic in typical

Quality Attribute	Explanation of Positive Effect
	service-oriented systems. Providing entity services and utility services in separate layers improves reuse by allowing business logic services across different domains to be implemented by composing entity and utility services as necessary. • Many changes to the data model can be handled by adapting the entity services, minimizing the impact on business logic services.
Reliability	Business process engines have built-in fault handling mechanisms. Besides, the business process workflow is executed strictly based on the business process model usually created using a BPM tool. Thus, a business process that uses a business process engine should be more reliable and less error-prone than a custom-developed workflow application.

4.5 Monitoring and Management Tools

Software monitoring and management tools are a group of tools that enable organizations to detect, diagnose, and react to potential problems in applications—these tools are often part of ESB products. SOA monitoring and management tools enable organizations to monitor and manage service-oriented systems. These tools are used for runtime monitoring to provide information that can be used to maintain system quality of service and to inform tactics and patterns such as

- Dynamic load balancing can use information from the monitoring tool about service response time or CPU and memory usage to distribute requests across replicated resources.
- Failover and recovery procedures can be activated based on failures reported by monitoring tools.
- Service removal or replacement can be triggered by failure detection and historical data on the rate of failure of a service in a particular configuration.

4.5.1 Supporting Patterns and Tactics

The set of tactics used for monitoring service-oriented systems includes

- **Heartbeat**: A heartbeat is a message sent periodically by the service to the monitoring component confirming that the service is running. A heartbeat does not verify that the service is working properly, only that it is responsive.
- **Ping**: In general, a ping is a message sent periodically by the monitoring component to the service that generates a response. A reply to the ping message within the expected response time indicates that the service is running and can process and respond to the ping.
- **Synthetic Transaction**: This is an extension of a ping using fabricated input to the service interface. A synthetic transaction verifies that a set of services is able to execute a transaction properly. This is often done using test accounts so that execution of these transactions does not impact business operations.
- **Status Message/Tracing**: This is a message that is sent based on the status of a transaction. Status messages enable application-level monitoring where services send detailed status messages at critical steps in a transaction, thus providing information for an end-to-end view of the transaction.
- **Agent**: An agent is an autonomous, continuously running software entity. Customized agents can monitor services, audit logs, memory probes, and incoming and outgoing messages, and report abnormal behavior such as a high request volume.

- **Instrumentation:** Instrumentation is code added to the service implementation to capture data related to metrics. The data (measures) is processed by monitoring tools, which can determine status and performance levels of the service.
- **Complex Event Processing (CEP):** With CEP, message traffic is monitored and analyzed in real-time to identify patterns of events that may be useful to the business or, in the case of service monitoring, may indicate a threat or an attack.

4.5.2 Impact on System Quality

Table 14 and Table 15 discuss the negative and positive effects in different quality attributes when a business process engine is used in a SOA solution.

Table 14: Monitoring and Management Tools Aspects that Negatively Affect System Qualities

Quality Attribute	Explanation of Negative Effect
Performance	There are many techniques (i.e., ping, heartbeat, and synthetic transactions) that are used for monitoring that consume system resources such as memory, processing cycles and network bandwidth. If not designed carefully, these monitoring messages may have a negative effect on the overall throughput.

Table 15: Monitoring and Management Tools Aspects that Positively Affect System Qualities

Quality Attribute	Explanation of Positive Effect
Availability	The monitoring tool can detect failures and notify the appropriate system artifacts to act in response to the failure (e.g., switching to a redundant software element or hardware).
Performance	Monitoring and management tools can be used to identify services that may be overloaded. New instances of service consumers can be dynamically created and the excess requests can be transparently routed to ensure proper load balancing.
Reliability	Synthetic transactions can be used to determine that services are accurately servicing requests or complying with policies.

5 Conclusions

Service-oriented architecture (SOA) is an architectural style for designing and developing distributed systems. Drivers for SOA adoption typically include easy and flexible integration with legacy systems, streamlined business processes, reduced costs, innovative service to customers, and agility to handle rapidly changing business processes. From an architectural and quality attribute perspective these drivers usually translate to interoperability and modifiability, which are achieved by adhering to a set of architectural principles for service-oriented systems such as loose coupling, standardization, reusability, composability, and discoverability. However, promoting interoperability and modifiability as well as adhering to these principles requires architects to make architectural decisions based on tradeoffs with other quality attributes that may be important to system stakeholders, or defined by SOA governance, such as availability, reliability, security, and performance.

Between 2005 and 2007, multiple surveys were conducted by organizations such as Forrester, Gartner, and IDC that showed that the top drivers for SOA adoption were mainly internally focused: these top drivers generally included application integration, data integration, and internal process improvement. This is changing. A recent survey published by Forrester shows that the number of organizations currently using SOA for external integration is approximately one third of the surveyed organization [Forrester 2009]. While the percentage of externally focused SOA applications is still a minority, this percentage has been growing and the trend will continue as organizations look at SOA adoption for supply-chain integration, access to real-time data, and cost reduction through the use of third-party services via the cloud or software-as-a-service (SaaS). As organizations expand their systems to cross organizational boundaries, architects will have to re-evaluate the use of SOA as an architectural style in these systems. They may need to architect their systems in such a way that qualities are met without having to sacrifice the loosely coupled, stateless, standards-based nature of the relationship between service consumers and service providers' characteristics that have made SOA a worthwhile technology to adopt.

In essence, as an architectural style, SOA may be an appropriate solution in some situations, but there will be other situations in which it is not appropriate or it has to be used in conjunction with other technologies to achieve the desired system qualities. The architect is often at conflict because on one hand there are business and mission goals that dictate the quality attributes that are important for system success. On the other hand, the architect using service-orientation wants to adhere to SOA principles and leverage SOA to its advantage and find out that it makes it difficult to achieve quality goals. The information provided in this report illustrates some of these conflicts and should help an architect navigate the underlying implications and provide reasons to be selective and deliberate in the architecting process. The architects of service-oriented systems play a crucial role in determining what expectations can or cannot be met by SOA adoption, and where tradeoffs can be made for the benefit of the organization and the accomplishment of system quality attributes.

References

URLs are valid as of the publication date of this document.

[Afshar 2007]
Afshar, M. *SOA Governance: Framework and Best Practices, Version 1.1*. Oracle, May 2007. http://www.oracle.com/us/technologies/soa/oracle-soa-governance-best-practice-066427.pdf

[AMQP 2010]
AMQP Working Group. Advanced Message Queuing Protocol 1.0 recommendation draft. http://www.amqp.org/ Accessed March 31, 2011.

[Arsanjani 2004]
Arsanjani, Ali. *Service-oriented modeling and architecture*. November 2004. http://www.ibm.com/developerworks/library/ws-soa-design1/

[Bass 2003]
Bass, Len; Clements, Paul; & Kazman, Rick. *Software Architecture in Practice*. Addison-Wesley Professional, 2003. http://www.sei.cmu.edu/library/abstracts/books/0321154959.cfm

[Bianco 2007]
Bianco, Phil; Kotermanski, Rick; & Merson, Paulo. *Evaluating a Service-Oriented Architecture* (CMU/SEI-2007-TR-015). Carnegie Mellon University, Software Engineering Institute, 2007. http://www.sei.cmu.edu/library/abstracts/reports/07tr015.cfm

[Bieberstein 2008]
Bieberstein, Norbert; Jones, Keith; Laird, Robert G.; & Mitra, Tilak. *Executing SOA: A Methodology for Service Modeling and Design.* July 2008. http://www.informit.com/articles/article.aspx?p=1194198

[Buschmann 1996]
Buschmann, F.; Meunier, R.; Rohnert, H.; Sommerlad, P.; & Stal, M. *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*. Wiley, 1996.

[Chappell 2004]
Chappell, D. *Enterprise Service Bus*. O'Reilly, June 2004 (ISBN 0-596-00675-6).

[Clements 2010]
Clements, Paul; Bachmann, Felix; Bass, Len; Garlan, David; Ivers, James; Little, Reed; Merson, Paulo; Nord, Robert; & Stafford, Judith A. *Documenting Software Architectures: Views and Beyond, 2nd Edition*. Addison-Wesley, 2010. http://www.sei.cmu.edu/library/abstracts/books/0321552687.cfm

[Erl 2008]
Erl, Thomas. *SOA: Principles of Service Design*. Prentice Hall, 2008 (ISBN: 0-13-234482-3).

[Erl 2009]
Erl, Thomas. *SOA Design Patterns*. Prentice Hall, 2009.

[Forrester 2009]
Forrester Research. *Enterprise and SMB Software Survey*, North America and Europe, Q4 2008, 2009.

[Gamma 1994]
Gamma, E.; Helm, R.; Johnson, R.; & Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[Ghosh 2000]
Ghosh, S. *Testing Component-Based Distributed Applications*. Purdue University, USA. 2000.

[Hohpe 2003]
Hohpe, Gregor & Woolf, Bobby. *Enterprise Integration Pattern*. Addison-Wesley, October 2003.

[Juric 2004]
Juric, M. B.; Kezmah, B.; Hericko, M.; Rozman, I.; & Vezocnik, I. "Java RMI, RMI tunneling and Web services comparison and performance analysis." *SIGPLAN Not. 39*, 5 (May 2004): 58-65. DOI= http://doi.acm.org/10.1145/997140.997146

[Lewis 2005]
Lewis, Grace & Wrage, Lutz. *A Process for Context-Based Technology Evaluation* (CMU/SEI 2005-TN-025). Carnegie Mellon University, Software Engineering Institute, 2005. http://www.sei.cmu.edu/library/abstracts/reports/05tn025.cfm

[Lewis 2007]
Lewis, Grace; Morris, Ed; Simanta, Soumya; & Wrage, Lutz. "Common Misconceptions about Service-Oriented Architectures." *Proceedings of the 6th IEEE International Conference on COTS-Based Software Systems (ICCBSS 2007)*, February 2007.

[Lewis 2008a]
Lewis, Grace, et. al. *Why Standards Are Not Enough To Guarantee End-to-End Interoperability*. Seventh International Conference on Composition-Based Software Systems (ICCBSS 2008), 2008, pp. 164-173.

[Lewis 2008b]
Lewis, G.; Morris, E.; Smith, D.; & Simanta, S. *SMART: Analyzing the Reuse Potential of Legacy Components in a Service-Oriented Architecture Environment* (CMU/SEI-2008-TN-008). Carnegie Mellon University, Software Engineering Institute, 2008. http://www.sei.cmu.edu/library/abstracts/reports/08tn008.cfm

[Morris 2010]
Morris, Ed; Anderson, Bill; Balasubramaniam, Sriram; Carney, David; Morley, John; Place, Pat; & Simanta, Soumya. *Testing in SOA Environments* (CMU/SEI-2010-TR-011). Carnegie Mellon University, Software Engineering Institute, 2010. http://www.sei.cmu.edu/library/abstracts/reports/10tr011.cfm

[OASIS 2006]
OASIS. *Web Services Security: SOAP Message Security 1.1* (WS-Security 2004). 2006. http://docs.oasis-open.org/wss/v1.1/

[OASIS 2007]
OASIS. *Web Services Reliable Messaging (WS-ReliableMessaging) Version 1.1*. http://docs.oasis-open.org/ws-rx/wsrm/200702/wsrm-1.1-spec-os-01.pdf

[OMG 2009a]
Object Management Group. *Business Process Model and Notation (BPMN) Version 2.0 Beta 1*. August 2009. http://www.omg.org/cgi-bin/doc?dtc/09-08-14

[OMG 2009b]
Object Management Group. *Service-Oriented Architecture Modeling Language (SoaML)—Specification for the UML Profile and Metamodel for Services (UPMS) Version 1.0 Beta 2*. December 2009. http://www.omg.org/spec/SoaML/1.0/Beta2/

[Simanta 2009]
Simanta, Soumya; Morris, Edwin J.; Lewis, Grace; Balasubramaniam, Sriram; & Smith, Dennis B. *A Scenario-Based Technique for Developing SOA Technical Governance* (CMU/SEI-2009-TN-009). Carnegie Mellon University, Software Engineering Institute, 2009. http://www.sei.cmu.edu/library/abstracts/reports/09tn009.cfm

[SOA Methodology 2010]
SOA Methodology. *SOA Methodology*. 2010. http://www.soamethodology.com

[Stomp 2010]
Stomp Protocol Specification, Version 1.0. 2010. http://stomp.codehaus.org/Protocol

[Sullivan 2009]
Sullivan, B. XML "Denial of Service Attacks and Defenses." *MSDN Magazine* (November 2009). http://msdn.microsoft.com/en-us/magazine/ee335713.aspx

[TOG 2006]
The Open Group (TOG). *The Open Group Architecture Framework (TOGAF).* 2006. http://www.opengroup.org/architecture/togaf8-doc/arch/index.html

[White 2004]
White, Stephen A. *Introduction to BPMN*. May 2004. http://www.bpmn.org/Documents/Introduction_to_BPMN.pdf

[WS-I 2010]
WS-I. *Web Services Interoperability Organization*. http://www.ws-i.org (2010).

[zur Muehlen 2008]
zur Muehlen, Michael. *How much BPMN do you need?* March 2008. http://www.bpm-research.com/2008/03/03/how-much-bpmn-do-you-need/

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
(Leave Blank)	August 2011	Final

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
Architecting Service-Oriented Systems	FA8721-05-C-0003

6. AUTHOR(S)

Philip Bianco, Grace A. Lewis, Paulo Merson, Soumya Simanta

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	CMU/SEI-2011-TN-008

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
HQ ESC/XPK 5 Eglin Street Hanscom AFB, MA 01731-2116	

11. SUPPLEMENTARY NOTES

12A DISTRIBUTION/AVAILABILITY STATEMENT	12B DISTRIBUTION CODE
Unclassified/Unlimited, DTIC, NTIS	

13. ABSTRACT (MAXIMUM 200 WORDS)

Service orientation is an approach to software systems development that has become a popular way to implement distributed, loosely coupled systems, because it offers such features as standardization, platform independence, well-defined interfaces, and tool support that enables legacy system integration. From a quality attribute point of view, the primary drivers for service orientation adoption are interoperability and modifiability. However, a common misconception is that an architecture that uses a service-oriented approach can achieve these qualities by simply putting together a set of vendor products that provide an infrastructure and then using this infrastructure to expose a set of reusable services to build systems. In reality, there are many architectural decisions that need to be made. An architectural decision that promotes interoperability or modifiability can negatively impact other qualities, such as availability, reliability, security and performance. The goal of this report is to present general guidelines for architecting service-oriented systems, how common service-oriented system components support these principles, and the effect that these principles and their implementation have on system quality attributes.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Service-oriented architecture, system architecture, quality attributes	46

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL